Лабораторная работа № 6
ПРОТОКОЛЫ ТРАНСПОРТНОГО УРОВНЯ

Цель работы

Изучить транспортные протоколы Интернет.

Приобрести практические навыки создания клиент-серверных приложений, взаимодействующих по протоколам TCP и UDP на основе применения классов среды .NET Framework.

Постановка задачи

1. Изучить методические указания к лабораторной работе, материалы лекций и рекомендуемую литературу.
2. Разработать приложения клиент-сервер для демонстрации работы с TCP и UDP протоколами согласно заданию.
3. Ответить на контрольные вопросы.

Методические указания

1. Протокол TCP

TCP (Transmission Control Protocol) – протокол управления передачей, один из самых широко распространенных протоколов транспортного уровня. Это ориентированный на соединение протокол, предназначенный для обеспечения надежной передачи данных между процессами. Описан в документах RFC 793, 1122, 1323.

TCP выполняет функции контроля ошибок и управления потоком данных.

TCP не поддерживает широковещание и многоадресную рассылку. Он может использоваться только для соединений «один-к-одному». При этом протокол устанавливает дуплексный виртуальный канал передачи между конечными узлами.

Каждый взаимодействующий процесс идентифицируется сокетом – парой IP-адрес и номер порта. Логическое соединение по протоколу TCP между двумя прикладными процессами идентифицируется парой сокетов. Каждый процесс одновременно может участвовать в нескольких соединениях.

Единицей данных TCP является сегмент.

2. Протокол UDP

UDP (User Datagram Protocol) - протокол пользовательских дейтаграмм.

Быстрый дейтаграммный протокол. Разработан для предоставления прикладным программам транспортных услуг.

Работает поверх IP. Не гарантирует доставку, не устанавливает виртуальных соединений, не осуществляет повторных передач, не выполняет переупорядочивания пакетов, не управляет потоком данных. Все эти функции при использовании UDP возложены на приложения (прикладные протоколы). Т.о. приложения должны сами обеспечивать надежность передачи.

По этим причинам UDP в основном служит для передачи мультимедийных данных, где важнее своевременность, а не надежность доставки.

Протокол UDP может работать по схеме «один-ко-многим», поэтому широко применяется для рассылки группового и широковещательного трафика.

3. Реализация протокола TCP на платформе .NET

Класс TcpListener

Для работы с TCP протоколом среда .NET Framework предоставляет высокоуровневые классы TcpListener и TcpClient, относящиеся к пространству имен System.Net.Sockets. В отличие от более низкоуровневого класса Socket, в котором при

получении/отправке данных применяется побайтовый подход, классы TcpListener и TcpClient используют потоковую модель. В них взаимодействие между клиентом и сервером основывается на потоке с применением класса NetworkStream. Они не поддерживают некоторые возможности, предлагаемые классом Socket, тем не менее, полезны во многих ситуациях.

Класс TcpListener обеспечивает работу с TCP-протоколом на стороне сервера. Он слушает запросы клиентов, принимает запрос и создает новый экземпляр класса Socket или TcpClient, который можно использовать для взаимодействия с клиентом.

Класс TcpListener инкапсулирует закрытый объект Socket, m_ServerSocket, доступный только для производных классов.

Основные методы класса TcpListener представлены в таблице.

Возвращаемый результат	**Название метода**	**Описание**
Socket	AcceptSocket()	Принимает соединение клиента и возвращает объект Socket, используемый для взаимодействия с клиентом
TcpClient	AcceptTcpClient()	Принимает соединение клиента и возвращает объект TcpClient, используемый для взаимодействия с клиентом
bool	Pending()	Показывает, есть ли запросы, ожидающие соединения
void	Start()/Start(int)	Переводит данный экземпляр TcpListener в режим прослушивания запросов соединения
void	Stop()	Закрывает данный экземпляр TcpListener, находящийся в режиме прослушивания

Основные свойства класса TcpListener представлены в таблице.

Тип	**Имя свойства**	**Описание**
IPEndPoint	LocalEndPoint	Открытое свойство LocalEndpoint возвращает объект IPEndpoint, который содержит информацию о локальном IP-адресе и номере порта, используемых для ожидания входящих запросов от клиентов.
bool	Active	Защищенное свойство, указывает, слушает ли в настоящий момент TcpListener запросы соединения.
Socket	Server	Защищенное свойство, возвращает базовый объект Socket, используемый объектом TcpListener, чтобы слушать запросы соединения.

Рассмотрим последовательность действий на стороне TCP-сервера.

1) Создание экземпляра класса TcpListener

Для создания экземпляра класса TcpListener (т.е. создания сокета) существуют три перегруженных конструктора:

1) public TcpListener(int point);
2) public TcpListener(IPEndPoint endPoint);
3) public TcpListener(IPAddress ipAddr, int port).

Первый конструктор указывает используемый для прослушивания запросов порт. При этом IP адрес равен IPAddress.Any, что эквивалентно значению 0.0.0.0. Т.е. сервер должен принимать сообщения клиентов на всех сетевых интерфейсах.

Во второй конструктор передается объект IPEndPoint, определяющий IP адрес и порт, на котором выполняется прослушивание.

Последний перегруженный конструктор принимает объект IPAddress и номер порта.

2) Прослушивание запросов клиентов

На следующем шаге после создания сокета можно приступать к прослушиванию запросов клиентов. Для этого в классе TcpListener есть метод Start(), выполняющий следующую последовательность действий:

1) Связывает сокет с IP адресом и портом, переданными в параметрах конструктору класса TcpListener (аналогично методу Bind() из класса Socket);

2) Вызывает метод Listen() базового класса Socket и начинает прослушивать запросы соединения от клиентов.

```
TcpListener tcpListener = new TcpListener(ipAddr, port);
tcpListener.Start();
```

Приступив к прослушиванию сокета, можно вызывать метод Pending(), чтобы проверять наличие ожидающих запросов соединения в очереди. Этот метод позволяет проверять наличие ожидающих клиентов до вызова синхронного метода Accept(), который блокирует выполняющийся поток.

```
if (tcpListener.Pending())
{       … В очереди есть запросы на соединение от клиентов    }
```

3) Прием соединений от клиентов

Для принятия запроса на соединение от клиента используются методы AcceptSocket() и AcceptTcpClient(). Они возвращают соответственно объекты Socket или TcpClient:

```
Socket sock = tcpListener.AcceptSocket();
TcpClient sock = tcpListener. AcceptTcpClient();
```

4) Отправка и получение сообщений

В зависимости от типа сокета, созданного при установлении соединения, обмен данными выполняется методами Send() и Receive() объекта Socket или с помощью чтения-записи объекта NetworkStream.

5) Остановка сервера

После завершения взаимодействия с клиентом нужно остановить слушающий сокет. Для этого используется метод tcpListener.Stop().

4. Класс TcpClient

Обеспечивает работу с TCP-протоколом на стороне клиента. Он построен на классе Socket и обеспечивает TCP-сервисы на более высоком уровне. В классе TcpClient есть закрытый объект данных m_ClientSocket, используемый для взаимодействия с сервером TCP.

Основные методы класса TcpClient представлены в таблице.

Название метода	Описание
Close()	Закрывает TCP-соединение
Connect()	Соединяется с удаленным хостом TCP
GetStream()	Возвращает объект NetworkStream, используемый для передачи данных между клиентом и удаленным хостом

Основные открытые свойства класса TcpClient представлены в таблице.

Тип	Имя свойства	Описание

LingerOption	LingerState	Устанавливает или возвращает объект LingerOption, содержащий информацию о том, будет ли соединение оставаться открытым после закрытия сокета и как долго
bool	NoDelay	Указывает, будет ли сокет задерживать отправку и получение данных, если буфер, назначенный для отправки или получения данных, не заполнен. Если свойство имеет значение false, TCP задержит отправку пакета, пока не будет накоплен достаточный объем данных. Это средство помогает избежать неэффективной отправки через сеть слишком маленьких пакетов.
int	ReceiveBufferSize	Задает размер в байтах буфера для входящих данных. Используется при считывании данных из сокета.
int	ReceiveTimeout	Задает время в миллисекундах, которое TcpClient будет ждать получения данных после инициирования этой операции. Если это время истечет, а данные не будут получены, возникнет исключение SocketException.
int	SendBufferSize	Задает размер в байтах буфера для исходящих данных.
int	SendTimeout	Задает время в миллисекундах, которое TcpClient будет ждать подтверждения числа байтов, отправленных удаленному хосту от базового сокета. При истечении времени SendTimeout порождается исключение SocketException.

Основные защищенные свойства класса TcpClient представлены в таблице.

Тип	Имя свойства	Описание
bool	Active	Указывает, есть ли активное соединение с удаленным хостом
Socket	Client	Задает базовый объект Socket, используемый объектом TcpClient. Поскольку это защищенное свойство, к базовому сокету можно обращаться, если класс произведен от TcpClient.

Создание сокета

Первый шаг при разработке TCP-клиента – создание экземпляра класса TcpClient.

В классе TcpClient определено четыре перегруженных конструктора:

1) public TcpClient();
2) public TcpClient(IPEndPoint ipEnd);
3) public TcpClient(string hostname, int port);
4) public TcpClient(AddressFamily family).

Если используется первый или второй конструктор, то для установления соединения с удаленным хостом надо вызывать метод Connect(), в котором указывать удаленную конечную точку, с которой надо соединиться.

Перегруженный конструктор TcpClient(IPEndPoint ipEnd) инициализирует новый экземпляр класса TcpClient, связанный с указанной локальной конечной точкой.

Перегруженный конструктор TcpClient(string hostname, int port) принимает в качестве параметров имя и порт удаленного узла. Он создает новый экземпляр класса TcpClient, разрешает указанное доменное имя в IP-адрес и устанавливает удаленное соединение с указанным хостом и портом. Однако с помощью этого конструктора нельзя задать локальный адрес и порт, с которым желательно связаться.

Еще один конструктор TcpClient(AddressFamily family) позволяет передать в качестве параметра значение перечисления AddressFamily.

Установление соединения с сервером

Следующий шаг – установление соединения с удаленным хостом. Для этого предназначен метод Connect().

Существуют несколько перегруженных вариантов метода Connect().

Имя	Описание
Connect(IPEndPoint)	Устанавливает соединение клиента с удаленным хостом, используя конечную точку IPEndPoint
Connect(IPAddress, int)	Устанавливает соединение клиента с удаленным хостом, используя указанный IP-адрес и номер порта
Connect(IPAddress[], int)	Устанавливает соединение клиента с удаленным хостом, используя несколько указанных в массиве IP-адресов и номер порта
Connect(String, int)	Устанавливает соединение клиента с удаленным хостом, используя имя хоста и номер порта

Так как установка соединения зависит от многих факторов: качества работы сети, наличия работающего приложения-сервера и т.д., рекомендуется всегда вызывать этот метод в блоке try/catch для обработки исключений.

Отправка и получение сообщений.

Данные передаются по сети между двумя узлами в форме непрерывного потока. Для обработки таких потоков в .NET имеется специальный класс NetworkStream.

Класс NetworkStream входит в пространство имен System.Net.Sockets и используется для отправки и получения данных через сокеты.

Объект NetworkStream – это небуферизованный поток, который не поддерживает произвольный доступ к данным. Невозможно изменить позицию внутри потока, и, следовательно, использование метода Seek() и свойства Position порождает исключение.

Прежде чем отправлять и получать данные, нужно определить базовый поток. Для этих целей в классе TcpClient есть метод GetStream(). Он создает экземпляр класса NetworkStream и возвращает его вызывающей программе

Следующий код создает экземпляр класса TcpClient, устанавливает соединение с сервером и затем создает экземпляр класса NetworkStream:

```
TcpClient client = new TcpClient();
client.Connect(“192.168.10.1”, 80);
NetworkStream tcpStream = client.GetStream();
```

Далее можно использовать методы Read() и Write() класса NetworkStream для чтения из потока и записи в поток.

Метод Write() используется для выполнения синхронной записи в поток. Он записывает в поток последовательность байтов и продвигает текущую позицию в потоке вперед на число записанных байтов.

Метод Write() принимает три параметра: массив байтов, содержащий отправляемые данные, позицию в потоке, с которой нужно начать запись, и длину данных.

```
tcpStream.Write(sendBytes, 0, sendBytes.Length);
```

sendBytes – отправляемый массив байтов.

Метод Read() используется для выполнения синхронного чтения из потока. Он считывает указанное число байтов и продвигает позицию в потоке вперед на число считанных байтов.

Метод Read() имеет такой же набор параметров – массив байтов для сохранения данных, которые считываются из потока, позицию начала считывания и число считываемых байтов.

```
byte[] rcvBytes = new byte[client.ReceiveBufferSize];
int length_rcvBytes = tcpStream.Read(rcvBytes, 0, client.ReceiveBufferSize);
```

Свойство ReceiveBufferSize класса TcpClient задает размер в байтах буфера для чтения, поэтому его можно использовать как размер массива байтов.

После взаимодействия с сервером, чтобы освободить все ресурсы и закрыть клиентский сокет, следует вызвать метод Close():

```
client.Close();
```

5. Реализация протокола UDP на платформе .NET

Класс UdpClient

Класс UdpClient предназначен для реализации в сети протокола UDP. Он построен на классе Socket.

Стандартная схема применения класса UdpClient такова. Необходимо создать экземпляр класса. Далее через вызов метода Connect() соединиться с удаленным хостом. В действительности метод Connect() в данном случае не устанавливает соединение, а служит для указания пункта назначения дейтаграммы. Оба шага можно сделать и в одной строке (без вызова метода Connect()), если сразу указать в конструкторе UdpClient удаленный адрес и порт.

Третий шаг – отправка и получение данных с помощью методов Send() и Receive(). Затем методом Close() закрывают сокет.

Одна из возможных схем отправки и получения данных по протоколу UDP с использованием класса UdpClient представлена на рисунке.

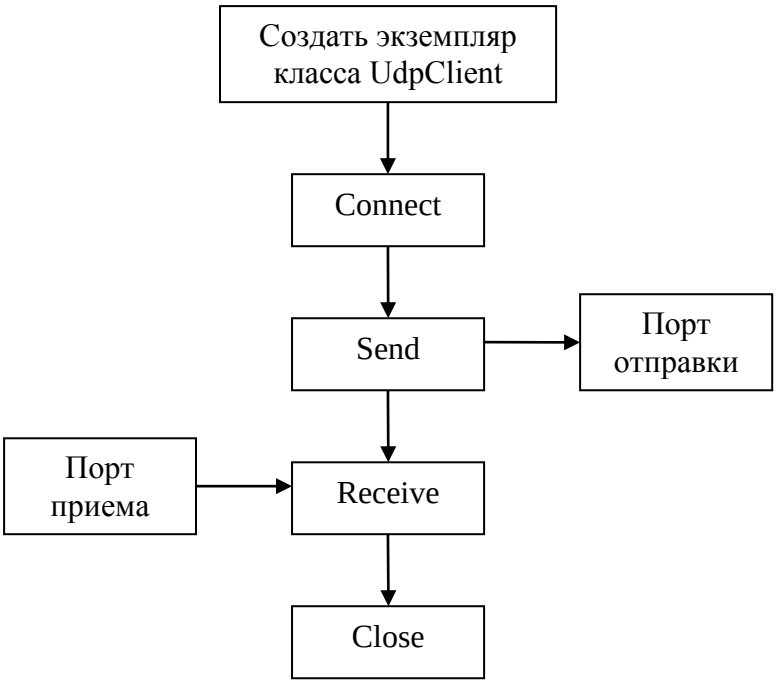

Рисунок - Схема отправки и получения данных по протоколу UDP

Экземпляр класса UdpClient можно создать несколькими способами, которые отличаются передаваемыми параметрами. Дальнейшее использование объекта UdpClient зависит от того, как он создавался.

Простейший способ состоит в вызове конструктора по умолчанию (без передачи параметров): UdpClient(). В этом случае далее нужно или вызвать метод Connect() или задать информацию о получателе при отправке данных в методе Send(). Если выбран конструктор по умолчанию, то при получении данных будут использоваться произвольный свободный порт и IP-адрес 0.0.0.0 .

Можно создать объект UdpClient, указав в качестве параметра только номер порта. В этом случае UdpClient будет слушать все локальные интерфейсы, т.е. опять использовать IP-адрес 0.0.0.0 .

Следующий способ создания объекта UdpClient заключается в использовании объекта IPEndPoint, который содержит локальный IP-адрес и номер порта. IP-адрес должен принадлежать одному из интерфейсов локальной машины, иначе произойдет ошибка.

Последний способ состоит в передаче конструктору UdpClient имени хоста и номера порта удаленного хоста. Это позволяет исключить шаг с вызовом метода Connect().

6. Многопоточное приложение клиент-сервер

В многопоточном сервере приложение работает в основном потоке, а для каждого подключившегося клиента создается новый поток. Поток существует все время взаимодействия с клиентом. При отключении клиента поток уничтожается.

В случае небольшого сервера и нескольких клиентов такой подход работает хорошо, к тому же его легко реализовать. К сожалению, многопоточный сервер плохо масштабируется. Главный его недостаток — большое число создаваемых и уничтожаемых потоков. Такой сервер может принять около 1000 соединений, после чего начинают генерироваться исключения, извещающие о нехватке памяти (память быстро заканчивается потому, что каждый поток имеет собственный стек, объем которого по умолчанию равен 1 МБ).

Избежать создания отдельного потока для каждого соединения можно создав заранее некоторое количество потоков - пул. При подключении клиента берется поток из пула потоков. В нем происходит взаимодействие с данным клиентом. Задачи выполняются параллельно. Если потоки не имеют общих данных, то не будет накладных расходов на синхронизацию, что делает работу достаточно быстрой. После завершения работы поток не убивается, а лежит в пуле, ожидая следующей задачи. Это убирает накладные расходы на создание и удаление потоков. Этот подход вполне приемлем, если соединения будут недолговечными. Однако соединения с серверами (например, с сервером-чатом) обычно остаются открытыми длительное время, поэтому ограничивать число соединений числом потоков в пуле зачастую неприемлемо.

Преимущества многопоточной модели: высокая скорость взаимодействия с каждым отдельным клиентом.

Недостатки многопоточной модели:

1) Каждый поток забирает часть ресурсов сервера. Следовательно, количество потоков ограничено мощностью сервера.

2) Большие накладные расходы на обработку потоков (создание, удаление, переключение между потоками).

3) Значительное время простоя процессора из-за невозможности перераспределения нагрузки между потоками.

Задания на лабораторную работу

Задание 1

1. Используя класс TcpClient, разработать консольное приложение TCP-клиент, устанавливающее удаленное соединение с сервером с использованием DNS-имени и номера порта. Клиент должен отправлять запрос серверу, получать ответ и отображать его для пользователя.

2. Используя класс TcpListener, разработать консольное приложение эхо-сервер для получения сообщений и отправки их назад TCP-клиенту. Сервер должен принимать сообщения клиентов по заданному локальному IP-адресу и номеру порта.

Задание 2

Разработать консольное приложение, осуществляющее взаимодействие по протоколу UDP. Использовать класс UdpClient. Приложение должно представлять собой простейший чат, позволяющий двум узлам, на которых оно запущено, обмениваться текстовыми сообщениями.

При запуске приложение должно запрашивать адреса локального и удаленного сокета.

Использовать отдельные потоки для получения и передачи данных, чтобы синхронные методы не заблокировали основной поток.

Задание 3 (Дополнительное)

Доработать TCP-сервер из задания 1, превратив его в синхронный многопоточный сервер. Для каждого подключающегося клиента создавать отдельный поток. Выводить на экран (или в файл журнала подключений) адреса подключившихся клиентов, время подключения, идентификатор обрабатывающего потока.

Продемонстрировать работу сервера, подключая большое количество клиентов в автоматическом режиме.

Контрольные вопросы

1. К какому уровню стека TCP/IP относится TCP-протокол и каковы его основные функции?
2. Как называется единица данных протокола TCP? Каков ее максимальный размер?
3. Как выполняется адресация приложений на транспортном уровне?
4. Как TCP устанавливает и разрывает соединение?
5. Какой алгоритм используется для обнаружения и исправления ошибок в протоколе TCP? Как он работает?
6. Перечислить достоинства и недостатки протокола UDP? В каких случаях применяется протокол UDP?
7. Какие классы в пространстве имен System.Net.Sockets .NET Framework обеспечивают поддержку сокетов и могут использоваться для работы с транспортными протоколами?
8. Какой конструктор класса TcpClient сам устанавливает удаленное соединение с сервером и не нуждается в вызове метода Connect? Какой конструктор класса TcpClient позволяет задать локальный адрес и порт, с которыми желательно связаться серверу?
9. Какими способами можно создать экземпляр класса UdpClient? Каким образом можно задать информацию о получателе дейтаграмм?
10. Каковы преимущества и недостатки многопоточной модели?